Metamorphic Domain-Specific Languages: A Journey Into the Shapes of a Language

Mathieu Acher, Benoit Combemale, Philippe Collet

Metamorphic Domain-Specific Languages

A Journey Into the Shapes of a Language

Mathieu Acher Benoit Combemale
University of Rennes 1, Inria, IRISA, France
mathieu.acher@irisa.fr, benoit.combemale@inria.fr

Philippe Collet
Univ. Nice - Sophia Antipolis, I3S, France
philippe.collet@unice.fr

Abstract

External or internal domain-specific languages (DSLs) or (fluent) APIs? Whoever you are – a developer or a user of a DSL – you usually have to choose side; you should not! What about metamorphic DSLs that change their shape according to your needs? Our 4-years journey of providing the "right" support (in the domain of feature modeling), led us to develop an external DSL, different shapes of an internal API, and maintain all these languages. A key insight is that there is no one-size-fits-all solution or no clear superiority of a solution compared to another. On the contrary, we found that it does make sense to continue the maintenance of an external and internal DSL. Based on our experience and on an analysis of the DSL engineering field, the vision that we foresee for the future of software languages is their ability to be self-adaptable to the most appropriate shape (including the corresponding integrated development environment) according to a particular usage or task. We call metamorphic DSL such a language, able to change from one shape to another shape.

Categories and Subject Descriptors D. Software [*D.2 SOFTWARE ENGINEERING*]: D.2.6 Programming Environments; D. Software [*D.3 PROGRAMMING LANGUAGES*]: D.3.0 General

Keywords programming; domain-specific languages; metamorphic

1. Introduction

Domain-specific languages (DSLs) are more and more used to leverage business or technical domain expertise within complex software-intensive systems. DSLs are usually simple and little languages, focused on a particular problem or aspect of a (software) system. Outstanding examples of DSLs are plentiful: Makefiles for building software, Matlab for numeric computations, Graphviz language for drawing graphs, or SQL (Structured Query Language) for databases.

DSLs are found to be valuable because a well-designed DSL can be much easier to use than a traditional library. The case of SQL is a typical example. Before SQL was conceived, querying and updating relational databases with the available programming languages led to a huge semantic gap between data and control processing. With SQL, users can write a query in terms of an implicit algebra without knowing the internal layout of a database. Users can also benefit from performance optimization: a query optimizer can determine the most efficient way to execute a given query.

Another benefit of DSLs is their capacity at improving communication with domain experts [1–3], thus tackling one of the hardest problems in software development. But DSLs are also ordinary languages, in the sense that many difficult design decisions must be taken during their construction and maintenance [1]. They usually can take different shapes: plain old to more fluent Application Programming Interface (APIs) ; internal or embedded DSLs written inside an existing host language ; external DSLs with their own syntax and domain-specific tooling. To keep it simple, a useful and common distinction is to consider that a DSL can come in two main shapes [2]: external or internal. When an API is primarily designed to be readable and to "*flow*", we also consider it as a DSL.

As for SQL – invented in 1974 and one of the first DSLs – it is interesting to note that it comes itself in different shapes. Figure 1 shows three of these shapes on the same basic query example. The top part of the figure shows the plain SQL variant, with a classical *"select, from, where"* clause. In the middle part of the figure, we show the same query written in Java with JOOQ (`http://jooq.org`), a fluent API that emphasises its typesafe nature. The lower part of the figure shows again the same query using the Slick API in Scala (`http://slick.typesafe.com`).

All shapes of a DSL have strengths and weaknesses whoever you are – a developer or user of the DSL. These SQL shapes illustrate this situation. The plain SQL version is an external DSL, making it easier for database experts to write complex queries, but making harder the software engineering job of integrating the DSL with other programming languages. On the entire other side, the JOOQ API is a Java internal DSL. As Java does not provide enough mechanisms to host DSL, the best result is a fluent API which mimics the SQL statement in successive method calls. Some of the SQL concepts are clearly recognizable by SQL experts, but some constructs, such as the AND clause, may lead to scoping errors. The job of the DSL developers is also reduced to an API implementation. The third example in Slick shows what can be achieved when the host language, here Scala, has some powerful constructs, such as "*filter*", that can be reused. The syntax is then less close to the original SQL, but easier for the average Scala developer. Another solution could have been to use the syntactic flexibility of Scala to closely mimic SQL, but this would not have suppressed some drawbacks, i.e., the internal DSL is a leaky abstraction: some arbitrary code may appear at different places in the domain scope [1, 2], and its concrete syntax can pose a problem for domain experts or non programmers [4].

The basic trade-offs between internal and external DSLs have been already identified and are subject to extensive discussions and research for several years. A new trend though is observed. DSLs are now so widespread that they are used by very different users with separate roles and varied objectives. In the case of SQL, users can be marketers, database experts,

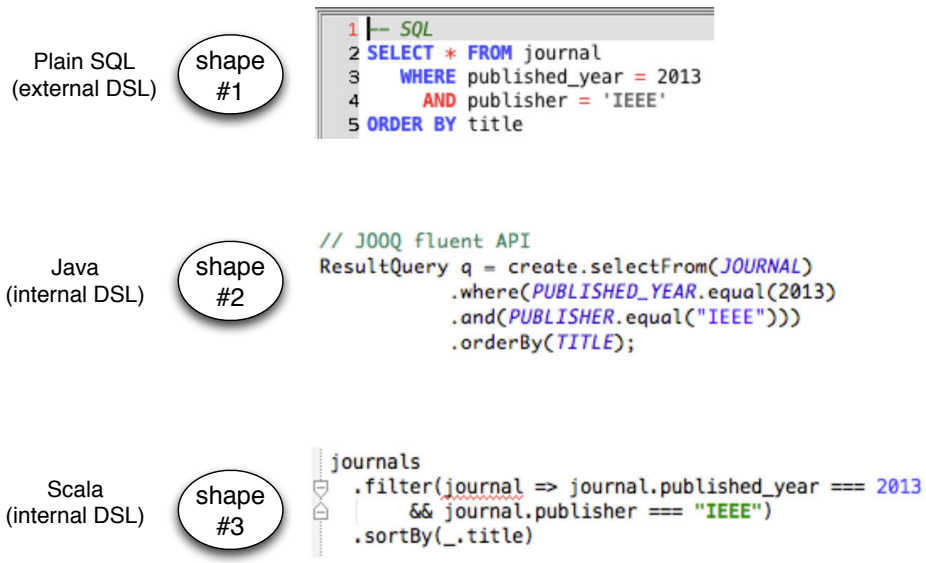

Figure 1. Three SQL *shapes*: plain SQL, JOOQ fluent API in Java, Slick API in Scala

system administrators, data warehouse managers, software engineers, or web developers. The objectives can vary a lot: prototyping of basic queries to search some data, sophisticated integration of SQL queries into a web application, etc. **Depending on the kinds of users, roles or objectives, an external shape or an internal shape of a language tends to be a better solution.** Most of the new learners of SQL (e.g., students) have started to learn SQL with explanations and examples written in the external shape of the language as well as a dedicated interactive environment. On the other hand, software engineers may have the need to use an internal shape when integrating database concerns.

This diversity poses a major challenge for the DSL engineering discipline:

> How to provide the good shape of a DSL according to the needs of a user?

The idea of having different shapes of a language is indeed appealing. In the case of SQL we can envision the use of different shapes and a transformation from one shape to another, for example, plain SQL could be converted to JOOQ code (and vice-versa). Users could rely on a familiar syntax, in their own environments; the integration of their work with other software ecosystems and languages could be facilitated as well. Yet it must be acknowledged that there is no concrete solution for realizing and supporting the idea.

One reason for the lack of solutions may be the complexity of the problem. By itself, developing one shape of a DSL with a dedicated syntax coming with a set of tools (e.g., type checkers, editing and refactoring facilities) is a difficult and time-consuming task [5]. Fortunately, the increasing maturity of language technologies (e.g., workbenches for creating external DSLs) has democratized the creation of numerous DSLs [2, 6–9]. Providing the support for transitioning from one shape to another is another difficult problem and open challenge. The number of bridges between N languages ($N \geq 2$) is theoretically exponential. Language workbenches such as Jetbrain's MPS support the transformation of a shape, but with the strong limitation of projecting in a fixed host language and/or environment.

Another barrier is the possible drawbacks of combining multiple programming languages (and multiple paradigms) in application development – sometimes called *polyglot programming* [10–12]. Yet we want to emphasize the fact that a user usually focuses on an unique shape – the most optimal one according to her task, know-how, education, or simply taste [13, 14]. It is the other stakeholders that are using different shapes in another context and environment.

Despite these socio-technical difficulties, our vision for the future of DSL engineering is as follows:

> The discipline, its foundations, methods, and tools, should go beyond the *constraints* which are imposed by the shapes that a DSL can take for its respective various users. We claim that developers and users of DSLs should not have to choose sides: a DSL should be *metamorphic* and change its shape accordingly!

Our vision is not only supported by an analysis of the DSL area (Section 2), but also grounded on practical experiences. SQL is not an isolated case of what a metamorphic DSL could be. We directly faced similar issues on building and evolving a DSL, called FAMILIAR [15]. Similarly to SQL, a diversity of shapes emerged and does make sense for a diversity of users, roles or objectives (Section 3). We hope that this essay will influence language designers and researchers so that they work at unifying the identified first steps towards DSLs that can change from one shape to another.

2. Shapes of DSL

We now discuss different socio-technical aspects of DSLs and how the idea of *metamorphic* DSLs came about w.r.t. these aspects.

2.1 API, Internal, External, Embedded, etc.

What is a DSL? A consensual, precise answer is hard to formulate. From a conceptual and very broad perspective, a DSL provides notations and constructs tailored toward a particu-

lar application domain. By trading generality for a focused support, DSLs promise substantial gains in productivity and ease of use in a limited domain. For instance, the Graphviz DSL provides a concise notation that makes the drawing of graphs easy and efficient.

DSLs are usually distinguished from *general*-purpose languages (GPLs). The distinction is useful to emphasize the restricted nature of DSLs and the specific abstractions they provide. In practice, the boundary between a GPL and a DSL is not as clear. For instance, a *"fluent"* API written in a GPL like Java, C# or Scala can be considered as a DSL. Not all APIs are DSLs; but when an API is primarily designed to be readable and to "*flow*", people[1] tend to consider it as a DSL. The reason is that fluent APIs (as DSLs) provide a convenient, concise notation and set of constructs to resolve specific problems in a domain. Another source of confusion for the boundary between GPL and DSL is the case of *embedded DSL*. For instance, in Lisp languages (GPLs), programmers can use macro systems to create new syntax and thus embed DSL.

The dichotomy between external and internal is generally employed to characterize DSLs. An *external* DSL is a completely separate language with its own custom syntax and environment. The domain-specific tooling includes editors (with syntax highlighting, auto-completion, etc.), debuggers, interpreters, compilers, etc. An *internal* DSL is a solution written on top of a host language (e.g., Java). Fluent APIs and embedded DSLs fall in this category. An internal DSL is limited to the syntax and structure of its host language. Some GPLs (e.g., Lisp languages) offer built-in facilities (e.g., macros) to develop and integrate new constructs. Extension mechanisms can also been developed to augment the syntax of a GPL [16] and embed DSLs. Both internal and external DSLs have strengths and weaknesses (learning curve, cost of building, programmer familiarity, communication with domain experts, mixing in the host language, strong expressiveness boundary, etc.) [2]. The case of SQL (see Figure 1) is quite representative of the phenomenon, and we will give another example in Section 3. But it is not the role of this essay to empirically investigate the superiority (if any) of a solution. We rather want to emphasize that a multiplicity of shapes does exist in practice.

The diversity of terminology shows the large spectrum of *shapes* DSLs can take. For example, these shapes include Fluent APIs, internal DSLs, (deeply) embedded DSLs, external DSLs, ...

2.2 Syntax and Environment Matter

Usability, productivity, learnability, expressiveness, reusability: all are factors developers and users of DSLs have to consider [17]. For instance, the users of a DSL have to learn an extra language, which takes time and effort. A very long learning curve may preclude the adoption of a DSL, despite the promise of a much better productivity. In fact, the adoption of a programming language, being a GPL or a DSL, is a very complex socio-technical problem [13, 14]. We argue that both the syntax and the supporting environment play a key role in this process.

Stefik *et al.* [4] investigated the role of syntax in what they call *the programming language wars* through four empirical studies. Ruby, Java, Perl, Python, Randomo, and Quorum (i.e., GPLs) have been considered. Results show that many aspects of traditional C-style syntax, while they have influenced a generation of programmers, exhibit problems in terms of usability for novices. Some languages are also considered more intuitive than others, by both programmers and non-programmers.

Denny *et al.* [18] conducted an empirical study showing that syntax in programming languages such as Java remains a major barrier to students. In an experiment involving 330 students in an introductory programming course, they found that even excellent students experience syntax issues. In [19] Denny *et al.* showed that the understanding of syntax errors varies as well.

The learnability of programming languages (e.g., Logo [20], Scheme [21], Smalltalk [22]), the usability of a language [23, 24], or the relationship between programming and natural languages [25] have also been investigated in the past. A number of authors have established that programming language usage impacts productivity in both the industry and open-source communities [26, 27]. At the paradigm level, Ramalingam and Wiedenbeck [28] compared comprehension with imperative and object-oriented styles. This study shows that language notations do influence novices. The authors suggest that language designers can exploit some of these findings to improve their languages.

In terms of programming language usability, a wide variety of topics have been studied, such as method naming [29, 30] coding standards [31], the use of identifiers [32], or API designs [33–37]. These topics are considered as well when designing an internal DSL (e.g., a fluent API). The choice of proper method names or identifiers, and the way of structuring them both contribute to the quality of a DSL syntax.

Another important aspect of a computer language (GPL or DSL) is the supporting tools. They can drastically influence human performance. No one now wants to (seriously) edit or program without advanced features offered by an *IDE (Integrated Development Environment)*, such syntax highlighting, code completion, debugging or refactoring services. DSLs can provide sophisticated static analyses and checks that go far beyond what can be done with GPLs. Even basic features (such as locating the source of error and presenting explanations) can be optimized thanks to the focus of a DSL. Eventually DSL users have to rely on specialized tools and dedicated environments to get the best of the language.

As syntax and development environment matter, IDEs should allow the user to choose the right shape of a DSL for each task.

2.3 Language Workbenches

Numerous approaches for implementing external or internal DSLs have been proposed.

Language workbenches have emerged as an efficient means to make the development of new DSLs affordable [2]. Developers can define, reuse and compose DSLs and obtain a comprehensive set of services, generally on top of an IDE.

A language workbench typically provides mechanism to define the concrete syntax of a DSL – being a textual or graphical notation or a combination thereof. Most language workbenches also provide syntactic and semantic editor services. Syntactic services include language-specific syntax coloring, outline views (for navigation support), folding facilities to hide part of a program, completion templates that suggest code, auto formatting of a program presentation, etc. Semantic services include semantics-preserving refactoring steps of programs, an error marker highlighting the source of error and presenting a message to the user, and quick fixes for cor-

[1] We use the term people to denote scientists (scholars), software professionals, bloggers, and any person interested in software development and languages.

recting an error. Language workbenches offer mechanisms to specialize the behavior of an editor, according to the specific syntax and semantics of a DSL.

With Spoofax [38] or Xtext [39], developers provide high-level specifications of a language while editor services are generated and integrated into an Eclipse environment. Projectional language workbenches such as Jetbrain's MPS support the projection of a representation into textual, symbolic, and tabular notations. MPS realizes the language-oriented programming paradigm introduced by Sergey Dmitriev [40]. For embedded software engineering, embeddr [6] provides an IDE and a set of integrated and extensible languages on top of the C language.

In addition, different strategies for embedding a DSL have been proposed [7, 16, 41, 42]. LMS proposes a staging mechanism to define an external DSL on top of the Scala language [43]. Tratt proposes a compile-time meta-programming facility to embed a DSL [7]. SugarJ demonstrates how to syntactically extend Java with a library of languages [16].

The community of language engineering is providing more and more mature solutions for building DSLs – being external or internal. Developers of DSLs have now a variety of strategies to choose from and build an appropriate shape.

2.4 Shaping up DSLs

An analysis of the language engineering era shows that the idea of having different shapes of a DSL is not so surprising:

- beyond the terminological clarifications and the ongoing debate of what is and what is not a DSL, we can observe that different shapes of a DSL have been characterized (internal, external, embedded DSLs, mini-languages, fluent API, etc.) by the community;
- the shape of a DSL is constituted by its syntax and its environment (e.g., IDE), two important factors for the adoption and success of a language;
- numerous tool-supported solutions exist to devise new shapes of DSL.

What is still missing is a systematic solution for transitioning from one shape to another. That is, we would like to open a given artefact (expressed in a DSL) with another syntax and another environment. In the next section, we will be more concrete to emphasize the argumentation and idea. We report on a practical experience representative of the multiplicity of shapes a DSL can take.

3. A Polymorphic Journey: The FAMILIAR experience

Over the past four years, we have continuously designed, developed, used, and maintained different solutions for *managing* feature models on a large scale. Feature models are by far the most popular notation in industry for variability modeling [44]. The formalism can be seen as a technology-independent, high-level, formal representation of options (features) and constraints of a configurable system (e.g., Linux [45]) or a family of products, also called software product lines [46].

As we will see, the FAMILIAR case study is representative of the "diversity" phenomenon we point out: multiple kinds of users, each with different objectives and needs, may potentially have to manage feature models. The purposes of feature models vary as well. Here is a non-exhaustive list of feature modeling users, together with an usage example:

- marketing engineers when characterizing the options (and their constraints) of products offered to customers;
- end users visualising feature models through an intuitive and interactive interface, a (Web) configurator;
- system engineers in charge of identifying common and reusable components;
- experts in model checking that use the feature modeling formalism for efficiently verifying complex properties of a configurable system;
- software engineers, for example, web developers in charge of deriving a web configurators from the feature model.

The diversity of usage impacts the important properties a solution should exhibit (e.g., learnability, expressiveness, reusability, usability, performance). Intuitively, the emphasis is likely to be more on usability and learnability for non programmers and on ease of integration and performance for software engineers. **Our story is that we developed "many shapes" of a language to mirror "many forms" of usage, making us question the nature of our solution.**

The story of FAMILIAR started at the beginning of 2010. At that time, the effort of researchers was mostly centered around the design of a graphical and textual language with a formal semantics for specifying feature models, and the development of *efficient* reasoning operations. On top of the feature modeling languages, numerous Java APIs arose for computing properties of the model. Naturally we looked at existing Java APIs (SPLAR, FeatureIDE, FAMA, etc.) that provide operations, based on different kinds of solvers (SAT, CSP, BDD). In terms of readability and learnability the APIs were rather complex to comprehend for a programmer (non expert in feature modeling) or a non programmer. The specific audience – people that want to develop efficient heuristics and have a fine-grained control over the internal details of the solvers – leads to an API design that did not fit all our requirements.

We aimed at providing a much more concise solution with less boilerplate, less technical details about the loaders and the solvers, and also more focused on the essential concepts users want to manipulate – feature models, configurations, and operations on the feature models themselves. In the meantime, we aimed at providing novel operations and a comprehensive environment for reverse engineering, composing, decomposing, and managing (in a broad sense) large-scale feature models.

#1 First try: FAMILIAR, an external DSL

We thus decided to move to a DSL. We created FAMILIAR [15] (for FeAture Model scrIpt Language for manIpulation and Automatic Reasoning) an external DSL with its own syntax and its own environment (editors, interactive console, etc.). Figure 2 gives an example of a FAMILIAR code, see shape #1. The first three lines almost implement the same behavior as the Java code written in SPLAR (see shape #0 in the same figure): loading of a feature model, counting of number of configurations and checking consistency properties. All the results are stored into variables (respectively fm1, n1, and b1). We also developed new facilities for supporting numerous formats and composing ("merge") or decomposing ("slice") feature models. Lines 4 to 6 of the FAMILIAR script load a new feature model (fm2) and compute two new models (fm3 and fm4).

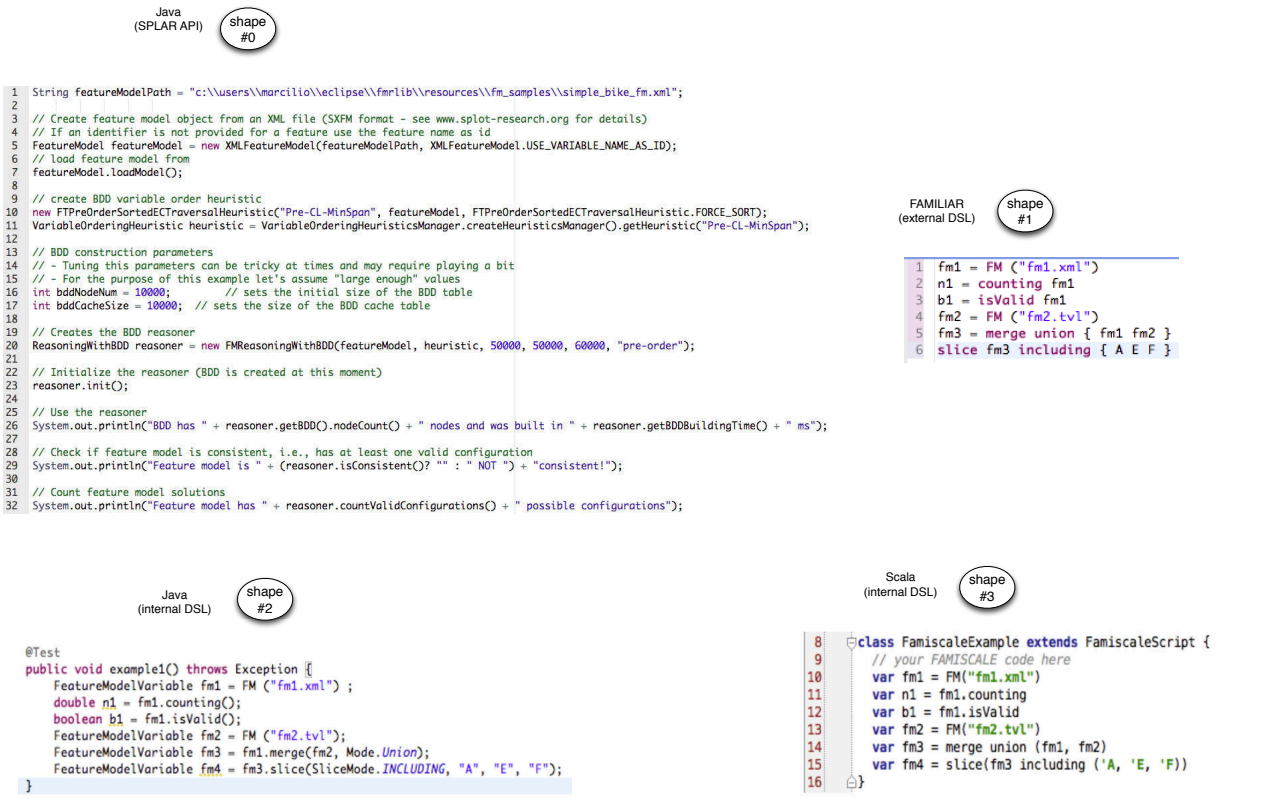

Figure 2. An example of SPLAR API and three polymorphic variants of the same domain-specific code snippet

The conciseness of the solution, the manipulation of specific abstractions and syntax, the hiding of solver details, the absence of boilerplate code, all are arguments in favor of a larger adoption. We report on some experiences hereafter to illustrate why the shift to a DSL appears to be beneficial. In the context of collaborative research projects in different application domains (military applications, video analysis, system engineering, web development) we collaborated and exchanged with experts usually unfamiliar with feature models and/or programming. Using FAMILIAR facilitated the communication during the meetings or brainstormings. Besides we taught to MSc students in Belgium and France an advanced introduction to feature modeling. Again using FAMILIAR facilitated the communication during the courses and lab sessions. The code we presented and discussed was only about feature modeling ; a well-defined vocabulary (e.g., counting, isValid) unifies the terminology and facilitates a common understanding. Students also experimented advanced aspects of feature modeling like the "merge" and "slice" with a specific environment. In both situations, users can interact with a read-eval-print loop (REPL) to have an immediate evaluation, possibly visual, of the result of the operations. Users can modify of the scripts, if needs be, and repeat the process. REPL and the scripting nature of FAMILIAR allow users to manually verify that the behavior is indeed what is expected, e.g., what have been understood for students.

We also used FAMILIAR code snippet into our scientific publications to ease our explanations (the vocabulary of the language is based on the terminology used by scientists) and to develop other advanced concepts. So far we cannot claim that FAMILIAR was crucial or simply better than existing solutions – evaluating the properties of a DSL is a very complex activity and a hot research topic *per se* we and others are currently investigating. But clearly, as industrial consultants and collaborators, teachers, or scientists, we identified advantages of using the external DSL and its dedicated environment.

#2 The (not so surprising) shift to a fluent Java API

So everything is done in a brave new world? Unfortunately no. The *external* nature of the DSL posed two kinds of problems.

First, the integration of FAMILIAR into complete applications was trickier than expected. For instance, we used FAMILIAR to model variability of video surveillance processing chains, with the ultimate goal of re-configuring the different vision algorithms according to contextual changes. Calling the FAMILIAR interpreter at runtime – each time a reconfiguration is needed – was clearly not a solution due to performance issues. Programming the "glue" between the adaptation logic, as provided by FAMILIAR, and the actual algorithms of the application was also a problem. We also faced integration problems when deploying configurable scientific services into the grid or when generating web configurators. We needed to operate over feature models and configurations, but in practice, the isolated nature of the external DSL leads to difficult or inefficient connections with the outside world and other software ecosystems (e.g., Java).

Second, the specific boundary of the language naturally limits its expressiveness. It is a desirable property of a DSL to restrict the users to essential language constructs and concepts of a domain. However there may be cons; and it was the case for FAMILIAR. Users wanted to iterate over a collection of feature models, e.g., to rename some features; to check some

properties and execute a specific operation, e.g., if the feature model is consistent, the "merging" with another is performed; to reuse some procedures, etc. We added to FAMILIAR some constructs of a general purpose language, like a "foreach"-like loop, an "if-then-else", and ways to define and call reusable scripts. We also added basic facilities for manipulating strings and integers – something offered for free by any general purpose language. Overall we found no elegant solution for comprehensively giving to users the right expressiveness, i.e., so that the language covers all feature modeling scenarios. And eventually we did not want FAMILIAR to resemble a general purpose language.

Due to the lack of integration and expressiveness of the external DSL in some situations, we developed a solution on top of a general purpose language. Specifically we designed a *fluent* Java API with the goal of being as close as possible to the initial syntax of FAMILIAR. Figure 2 gives the corresponding Java code of the initial FAMILIAR script (see shape #2). The API has the merit of being concise and provides idiomatic facilities for loading a feature model and executing reasoning operations. But compared to FAMILIAR scripts, we identified the following drawbacks. First, the distance with the original syntax leads to less concise code and more boilerplate code. Second, the boundary of the proposed solution is less rigid; domain-specific abstractions are as accessible as internal details of solvers or constructs of a general purpose language. For instance, users run the risk of developing inefficient operations instead of reusing existing ones. It may be relevant for some kinds of users but the experience can be disconcerting for others, e.g., new learners such as students. Third, no specific environment, e.g., with REPL and graphical editors, has been developed, thus making complex the realization of scenarios in which users interactively play with the models and the operations.

#3 Yet another attempt: going internal with FAMISCALE

To overcome some of the previous limitations, we then aimed at reducing the gap with the original FAMILIAR syntax while preserving some integration capabilities with a general-purpose programming language. We thus developed yet another solution (another internal DSL). We build FAMISCALE on top of the Scala language that notably supports a flexible syntax, implicit type conversions, call-by-name parameters and mixin-class composition. Compared to the Java solution, the gap with FAMILIAR is largely reduced while the Scala interactive interpreter and integration facilities are directly reusable.

The corresponding FAMISCALE code of our feature model example is depicted in the lower part of Figure 2 (see shape #3). It clearly shows the nearness with the original FAMILIAR syntax, but still, one cannot directly interpret some FAMILIAR code. During some recent application, we faced the need to compute some additional metrics on feature models. Coding and integrating them in the software toolchain was very fast and the newly developed functionalities could be added to a future release seamlessly. On the contrary, using the advanced concepts of the Scala host language was a considerable engineering effort and the whole solution is not that close to FAMILIAR. For example handling variable namespaces and parameterized scripts led to a very different solution from FAMILIAR, but with the benefit of a better integration with the programming side of the Scala platform.

All wrong? No. Metamorphic!

So what is the best solution? The fluent Java API? The scripting language (FAMILIAR)? The internal DSL in Scala (FAMISCALE)? We considered various tradeoffs, e.g., learnability, expressiveness, reusability, throughout our exploration journey. Our experience is that there is no one-size-fits-all solution or clear superiority of a solution compared to another. We have a better expressiveness with a Java API, but a more difficult solution to apprehend. We have a better interactive environment with FAMILIAR but we cannot realize all feature modeling scenarios – we even doubted at certain points of our development that FAMILIAR *is* a DSL.

So, are we all wrong from the start? We may have missed a solution that outperforms all the others; or a future programming technology might emerge to develop such a solution. Yet, our experience is that it does make sense to continue the maintenance of external and internal solutions. Rather than choosing between an internal DSL or an external DSL, we argue that both solutions are eventually relevant.

For instance, new learners of feature models benefit from playing with a specific environment and a dedicated, epurated syntax (FAMILIAR). Software engineers make a far better use of the internal DSL to integrate the feature modeling logic as part of their software project. Why forcing a software engineer to use an external DSL? or a student to start with a Java API? Even for a specific kind of user, the different shapes of the language are likely to be useful. For instance, software engineers themselves can prototype a scripting solution with the external DSL, play with the FAMILIAR environment, and eventually get back to the "internal" shape. Different stakeholders can also be part of the process while using different shapes of the solution (see Figure 3). A marketing engineer will more likely use the external DSL to characterize the variability of the system under design, to control some properties of the feature model, etc. Once achieved, the software engineer in charge of connecting the different feature models will go on with the development.

We argue that this situation and scenarios are generalisable in many DSL engineering contexts, beyond SQL and FAMILIAR. Our key point is that **the "best" shape of a language heavily depends on the usage context (tasks to perform, kinds of users, etc.): in the FAMILIAR experience, any existing shape we developed has, at some points, some qualities that others do not**.

4. Metamorphic DSL: A Vision For the Future of Languages

Based on our own experience and the analysis of the time-honored evolution of computer languages, we now show how this vision of metamorphic DSLs finds its roots in ongoing work, how it can be defined and what challenges it brings to the community.

Ongoing Work. As we have analyzed in Section 2, some recent work already started to wipe off the gap between the different shapes. For example, some approaches for implementing external DSLs try to incorporate the benefits from internal DSL by supporting the reuse of already implemented languages and tools. For instance, Xtext supports the reuse of language libraries such as Xbase, which come with their whole tooling such as editor, type checker and compiler [39]. Conversely, some approaches try to limit the scope of the host infrastructure that can be reused in an internal DSL to ensure a well-defined isolation of the domain-specific constructs.

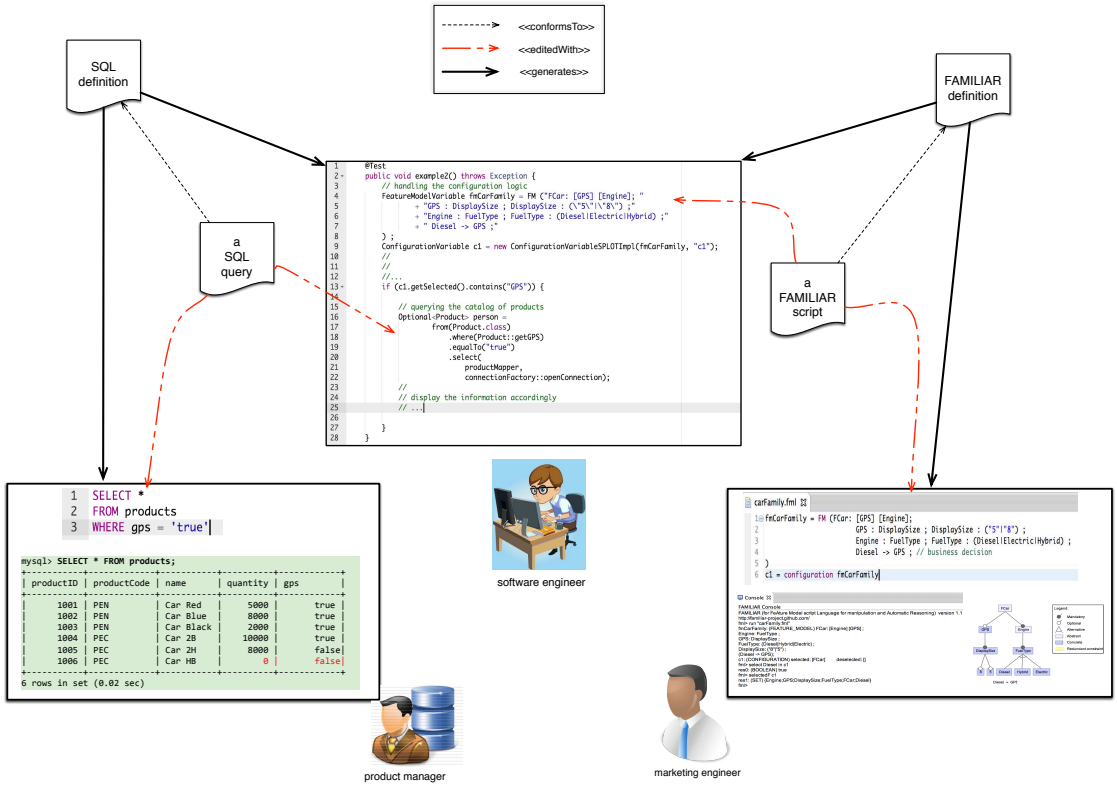

Figure 3. Concrete illustration of the vision for integrating a metamorphic SQL and a metamorphic FAMILIAR. Java allows one to generate an IDE for integrating the two languages

For example, LMS relies on the staging mechanism to define an external DSL on top of the Scala host infrastructure [43]. Besides, projectional language workbenches such as Jetbrain's MPS support the projection in a shape of a purely external DSL or as embedded in a host language similarly to internal DSLs [6]. More generally different strategies for embedding a DSL have been proposed [7, 16, 41, 42].

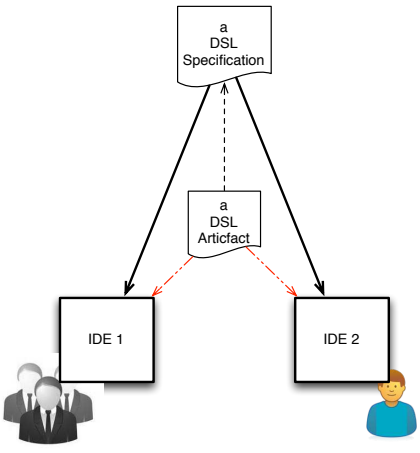

Figure 4. Metamorphic DSL (abstract scenario)

These recent efforts attempt to integrate the advantages of the different approaches, progressively bridging the gap between them. This is an essential experience to master the various possible bridges and differences between the different shapes. Nevertheless, the same concern (variability or database queries in our example) usually flows through the life cycle, being addressed by different stakeholders with their specific points of view and objectives. Each user expects to manipulate the same programming artefact through the most appropriate shape of the DSL (incl., the whole tooling). For example, a product manager would manipulate SQL queries with plain text SQL and dedicated tools while a software engineer would use an internal DSL (see Figure 3 for an illustrative scenario).

Vision. Beyond the unification of the different approaches, the vision that we foresee is

> the ability of software languages to be self-adaptable to the most appropriate shape (including the corresponding IDE) and according to a particular usage or task. We call *metamorphic DSL* such a domain-specific language, able to change from one shape to another.

From the same language description, we envision the ability to derive various IDEs that can be used accordingly (see Figure 4). The challenge consists in supporting the manipulation of the same artefact from the different IDEs dedicated to the different points of view, each one with their specific representation as well as integration into a host infrastructure.

In our vision, the same programming artefact could be started in isolation so that a stakeholder could describe her concern with a highly dedicated environment. The same artefact would flow (e.g., refinement, transformation, composi-

tion, consistency checking) and be combined to the other concerns until eventually obtaining the final global system. The vision we propose does not conflict with the use of multiple languages [47]. It is rather a way to support their *integration* for a coordinated development of diverse domain-specific concerns of a system.

The scenario of Figure 3 illustrates the vision on two DSLs, namely SQL and FAMILIAR. It involves different stakeholders (marketing engineer, product manager, software engineer) that aim at providing a configurator of sales product. Each DSL provides two IDEs from the same language definition (incl., one shared by the two DSLs) that can be indifferently used to edit the same conforming artefact. A shared IDE is used as a common infrastructure to integrate the two concerns (e.g., the integration of FAMILIAR and SQL in Java).

Challenges. The integration of multiple metamorphic DSLs raises many challenges. One must still find solutions for the integration of domains, especially between business and technical domains. Systematic methods for evaluating *when* a shape of a DSL meets the expected properties (e.g., learnability) and is more suitable to another would benefit to developers of DSL, but are far from being complete. The vision also gives rise to questions about the modularization of artefacts. The technical challenge is to share some information, while being able to visualize and manipulate an artefact in a particular representation and in a particular IDE. A global mechanism must ensure consistency of the artefacts between these heterogeneous IDEs.

The ability to shape up a DSL would open a new path for an effective communication between humans and the realization of global software engineering scenarios: could Metamorphic DSLs bring language engineering to the next level, enabling user-driven task-specific support in domain-specific worlds?

Acknowledgments

We thank our colleagues Thomas Degueule, Guillaume Bécan, Olivier Barais, Julien Richard-Foy, and Jean-Marc Jézéquel for fruitful comments and discussions on earlier drafts of this paper. We also thank Jonathan Aldrich for his advices and remarks. This work is part of the GEMOC Initiative, and partially funded by the ANR INS Project GEMOC (ANR-12-INSE-0011).

References

[1] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[2] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 2010.

[3] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[4] Andreas Stefik and Susanna Siebert. An Empirical Investigation into Programming Language Syntax. *Trans. Comput. Educ.*, 13(4):19:1–19:40, November 2013.

[5] Steven Kelly and Risto Pohjonen. Worst Practices for Domain-Specific Modeling. *IEEE Software*, 26(4):22–29, 2009.

[6] Markus Voelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schätz. mbeddr: instantiating a language workbench in the embedded software domain. *Autom. Softw. Eng.*, 20(3):339–390, 2013.

[7] Laurence Tratt. Domain Specific Language Implementation via Compile-time Meta-programming. *ACM Trans. Program. Lang. Syst.*, 30(6):31:1–31:40, October 2008.

[8] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge. In *SLE*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer, 2013.

[9] Steven Kelly, Kalle Lyytinen, Matti Rossi, and Juha-Pekka Tolvanen. MetaEdit+ at the Age of 20. In *Seminal Contributions to Information Systems Engineering*, pages 131–137. Springer, 2013.

[10] Dean Wampler. Polyglot programming `http://www.polyglotprogramming.com/`.

[11] Bertrand Meyer. Multi-language programming: how .NET does it. *Software Development (3-part article Part 1: Polyglot Programming; Part 2: Respecting other object models; Part 3: Interoperability: at what cost, and with whom?)*, May, June and July 2002.

[12] Martin Fowler. One language `http://martinfowler.com/bliki/OneLanguage.html`, 2007.

[13] Leo A. Meyerovich and Ariel S. Rabkin. Socio-PLT: Principles for Programming Language Adoption. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '12, pages 39–54. ACM, 2012.

[14] Leo A Meyerovich and Ariel S Rabkin. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 1–18. ACM, 2013.

[15] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming (SCP)*, 78(6):657–681, 2013.

[16] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: Library-based syntactic language extensibility. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 391–406. ACM, 2011.

[17] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Domain-Specific Languages in Practice: A User Study on the Success Factors. In *Model Driven Engineering Languages and Systems*, pages 423–437. 2009.

[18] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. Understanding the syntax barrier for novices. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, pages 208–212. ACM, 2011.

[19] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. All syntax errors are not equal. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '12, pages 75–80. ACM, 2012.

[20] George Lukas. Uses of the LOGO Programming Language in Undergraduate Instruction. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, pages 1130–1136. ACM, 1972.

[21] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A Programming Environment for Scheme. *J. Funct. Program.*, 12(2):159–182, March 2002.

[22] Alan Borning and Tim O'Shea. Deltatalk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk-80 Language. In *European Conference on Object-oriented Programming on ECOOP '87*, pages 1–10. Springer-Verlag, 1987.

[23] L.K. McIver, Monash University. School of Computer Science, and Software Engineering. *Syntactic and Semantic Issues in Introductory Programming Education*. Monash University, 2001.

[24] John F. Pane, Brad A. Myers, and Chotirat Ann Ratanamahatana. Studying the language and structure in non-programmers' solutions to programming problems. *Int. J. Hum.-Comput. Stud.*, 54(2):237–264, February 2001.

[25] D. P. Delorey, C. D. Knutson, and M. Davies. Mining programming language vocabularies from source code. In *21st Annual Psychology of Programming Interest Group Conference - PPIG*, 2009.

[26] Craig Comstock, Zhizhong Jiang, and Peter Naudé. Strategic software development: Productivity comparisons of general development programs. *World Academy of Science, Engineering and Technology*, 34:25–30, 2007.

[27] Daniel P. Delorey, Charles D. Knutson, and Scott Chun. Do Programming Languages Affect Productivity? A Case Study Using Data from Open Source Projects. In *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*, FLOSS '07, pages 8–12. IEEE Computer Society, 2007.

[28] Vennila Ramalingam and Susan Wiedenbeck. An Empirical Study of Novice Program Comprehension in the Imperative and Object-oriented Styles. In *Papers Presented at the Seventh Workshop on Empirical Studies of Programmers*, ESP '97, pages 124–139. ACM, 1997.

[29] Einar W. Høst. Understanding programmer language. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 943–944. ACM, 2007.

[30] Einar W. Høst and Bjarte M. Østvold. The programmer's lexicon, volume i: The verbs. In *SCAM*, pages 193–202. IEEE, 2007.

[31] David Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. To camelcase or under_score. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 158–167. IEEE, 2009.

[32] Florian Deissenboeck and Markus Pizka. Concise and consistent naming. *Software Quality Control*, 14(3):261–282, September 2006.

[33] Jeffrey Stylos and Steven Clarke. Usability Implications of Requiring Parameters in Objects' Constructors. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 529–539. IEEE Computer Society, 2007.

[34] Jeffrey Stylos and Brad A. Myers. The Implications of Method Placement on API Learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 105–112. ACM, 2008.

[35] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. What Should Developers Be Aware Of? An Empirical Study on the Directives of API Documentation. *Empirical Software Engineering*, 17(6):703–737, 2012.

[36] Steven Clarke. Measuring API usability. *Dr. Dobb's Journal*, 29:S6–S9, 2004.

[37] Brian Ellis, Jeffrey Stylos, and Brad Myers. The Factory Pattern in API Design: A Usability Evaluation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 302–312. IEEE Computer Society, 2007.

[38] Lennart C.L. Kats and Eelco Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 444–463. ACM, 2010.

[39] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: Implementing Domain-specific Languages for Java. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, pages 112–121. ACM, 2012.

[40] Sergey Dmitriev. Language oriented programming: The next programming paradigm. Technical report, JetBrains, 2004.

[41] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of dsls. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, GPCE '08, pages 137–148. ACM, 2008.

[42] Andy Gill. Domain-specific Languages and Code Synthesis Using Haskell. *ACM Queue*, 12(4):30:30–30:43, April 2014.

[43] Tiark Rompf and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. *SIGPLAN Not.*, 46(2):127–136, October 2010.

[44] David Benavides, Sergio Segura, and Antonio Ruiz Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, 2010.

[45] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. A study of variability models and languages in the systems software domain. *IEEE Transactions on Software Engineering*, 39(12):1611–1640, 2013.

[46] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer-Verlag, 2013.

[47] Benoit Combemale, Julien Deantoni, Benoit Baudry, Robert France, Jean-Marc Jézéquel, and Jeff Gray. Globalizing Modeling Languages. *Computer*, pages 68–71, June 2014.